View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Syracuse University Research Facility and Collaborative Environment

Syracuse University

SURFACE

Center for Advanced Systems and Engineering College of Engineering and Computer Science

1992

Flattening C++ Classes

Umesh Bellur
Syracuse University, CASE Center

Al Villarica
Syracuse University, CASE Center

Kevin Shank
Syracuse University, CASE Center

Imram Bashir
Syracuse University, CASE Center

Doug Lea
SUNY Oswego

Follow this and additional works at: https://surface.syr.edu/case

 Part of the Programming Languages and Compilers Commons

Recommended Citation

Bellur, Umesh; Villarica, Al; Shank, Kevin; Bashir, Imram; and Lea, Doug, "Flattening C++ Classes" (1992). *Center for Advanced Systems and Engineering*. 1.
https://surface.syr.edu/case/1

This Report is brought to you for free and open access by the College of Engineering and Computer Science at SURFACE. It has been accepted for inclusion in Center for Advanced Systems and Engineering by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

Flattening C++ Classes*

Umesh Bellur Al Villarica Kevin Shank Imram Bashir
Doug Lea
New York CASE Center, Syracuse NY 13244

August 21, 1992

Abstract

Inheritance with derived classes and virtual functions are key design concepts in C++. Despite this, their use can result in significant degradation of run time performance. We present a class flattening tool, which we believe will help eliminate the overhead associated with virtual functions in C++ programs. A flattener may also prove useful in the reuse, debugging, and understanding of C++ components. This report deals with the issues associated with flattening, and then presents a detailed design of such a tool.

*This work has been sponsored in part by a grant from Hewlett Packard to New York state CASE Center. This report has also been submitted as NY CASE Center TR-92-23

Contents

1 Introduction

This report discusses a tool addressing with two important issues:

- Run time performance problems of C++ programs arising out of the use of virtual functions and inheritance.
- Reuse in programming environments.

As one solution to problems in both these areas, we present a *class flattener* – a tool that helps eliminate the class hierarchy in C++ programs by repackaging derived classes as though they were base classes.

The main goal of this project is to build a stand alone facility which could be integrated into a complete C++ program development environment such as the HP SoftBench. Two possible benefits of this tool are:

1. To improve run time performance of C++ programs. In this situation the flattener will act as a filter between the preprocessor and the C++ compiler. The developer can selectively flatten code to optimize the program for efficiency.
2. To act as a useful option of a "class browsing" facility.

Additional constraints may be placed in order to maximize usability:

- It should be platform (machine and compiler) independent so that it can be made available to a wide range of users.
- The tool itself should be optimized for efficiency and should take up a minimum of resources and time.
- It should be easy to use and should output clear, understandable code.

The position of the flattener in the usual process of code development is shown in figure 1. It serves as a filter either between (1) the preprocessed C++ code and the compiler, when used to optimize code, or (2) between the preprocessed code and the display when used as an option in a browser facility.

One of the two approaches to flattening described in this technical report (the parallel hierarchy method) does not aid in class browsing. It only helps improve the execution performance. Conversely, the other method is aimed at generating easily readable code, but cannot include support for the subtleties of C++ subclassing and function resolution rules without semantic assistance from a compiler or other tool. As discussed below, we have found that these designs have complementary strengths and weaknesses. And to preview a conclusion, on the basis of our experience, it appears most productive to specialize tools primarily addressing these separate roles.

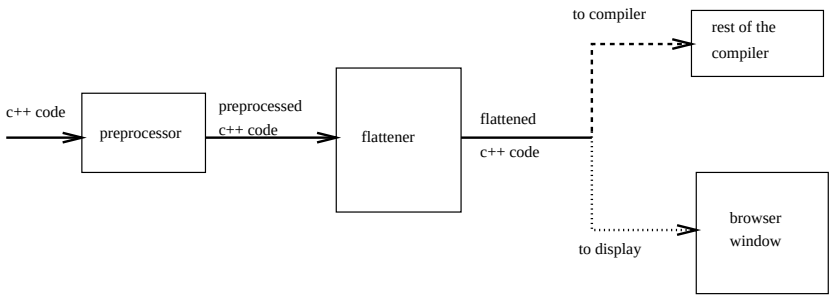

Figure 1: An overview of the process

1.1 Related Work

There has not been much work directly related to C++ class flattening for the purpose of achieving better performance. One approach was that of [1] wherein the authors accomplish flattening by merging header files. But this approach was limited in functionality to exactly one level of inheritance, and the classes were written to conform to the expectations of the tool.

Several existing browsers (e.g., [2]) include limited flattening capabilities that indicate the presence of parent declarations, without actually reconstituting code.

1.2 Overview

The remainder of this report is organized as follows. In the next section we discuss why and how flattening might be used, and present a high-level summary of desired functionality. Following that we discuss in detail the design issues involved in constructing a class flattener. We then report a summary of current status with a protype tool and provide recommendations about future work.

2 Functionality

2.1 Performance

Inheritance and virtual functions are powerful mechanisms in any object oriented programming language. Virtual functions enable polymorphism and enhance functionality via inheritance and code sharing. However, the power delivered by these features is associated with a certain cost.

Virtual functions are typically implemented as indirect function calls. The use of indirection adds some small additional overhead to the normal cost required for non-virtual methods and free-standing functions. When virtual functions perform nontrivial computations, this cost becomes insignificant. However, for "lightweight" functions, like those that merely return the value of a member variable, two problems emerge [3]:

- The calling overhead overwhelms computation time.
- Standard "backend" compiler optimizations are not possible, since the values of returned variables and simple computations are not "visible" to the compiler, and so cannot be simplified, removed as redundant, or procedurally integrated.

Dealing with these issues are the best reasons for programmers to declare member functions as `inline`. The effects of inlining are *extremely* context dependent. In the worst cases, inlining may waste tremendous amounts of code space and (often as a result of this) even slow down execution. In the best cases, actual space *reductions* and order-of-magnitude speedups have been observed, mainly because of the downstream optimizations that inlining enables. Because the effects are so application dependent, programmers themselves are usually the best judges of when to use inlining.

However, because of the indirection involved in virtual function calls, existing C++ compilers cannot inline expand member functions declared as both `inline` and `virtual` except perhaps in special situations. Thus programmers are not able to tune programs involving virtual functions via selective inlining.

The simplest characterization of a flattener is that it is a tool that transforms code so that more `virtual inlines` may actually be inlined.

Indeed, in order to support this kind of tuning, *compilers* would need to employ automated versions of the strategy used for the present compiler-independent tool. Chambers and Ungar [4] have shown the utility of this approach (a form of "customization") in a compiler for the object oriented language Self.

2.1.1 Functional Description

The general technique of flattening for performance is easy to describe. A flattener accepts as input a program, along with the name of a class `K` (re)declaring or inheriting at least some virtual methods from some superclass `superK`, as well as object declarations that are indicated as flattenable (either through arguments to the tool or annotations

within the program text). It outputs (typically via a "pipe" to a C++ compiler) a revised program identical to the input program, except

1. For every class `K` indicated, a new class `FlatK` is added. `FlatK` is identical to `K` except that no methods are virtual. (As described in below, other classes and methods may be added as well in order to obtain this effect.)
2. All *indicated* variable and object declarations of type `K` are changed to be of type `FlatK`.
3. For each function `f` that could accept an argument of type `K` (as nominally indicated via `K` or any ancestor thereof), a new version of `f` is created to accept type `FlatK`.

Thus, a downstream compiler will "see" `flatK`, with all functions non-virtual (and possibly inlinable). The potential performance benefits come at the expense of the added specialized versions of class and client code. Because both the original and flattened versions of classes will exist in a program, all cases in which objects are not transformed into flattened versions will continue to work, although not as efficiently as those occurrences that were flattened.

A more extensive tool would *automatically* detect objects that are known by the tool to be safe for flattening. A declaration is safe if the object is never rebound to a variable of a superclass type that is in turn sent to a function in which virtual methods are invoked. (It is exactly and only this usage that absolutely requires support for dynamic resolution of virtual functions.)

As described in more detail below. a full assessment of safety would require full program-wide dataflow analysis. This is not a practical approach for such a tool, but becomes possible if flattening were to become part of an optimization suite built within a compiler. While this would be very desirable, simple strategies suffice to construct a useful tool. The tool should typically be used to avoid particular performance problems, so manual intervention is effective.

The mere existence of a flattener tool can have a liberating effect on C++ class design. Among other things, it encourages the good practice of declaring *all* members as `virtual` unless there are conceptual reasons (not performance reasons) to do otherwise. In those cases where full polymorphism is not needed, a flattener may remove performance penalties.

In particular, flattening is especially useful in helping to remove performance penalties in programs that make heavy use of *abstract classes* (classes defining interfaces in terms of "pure virtual" functions). Most usages of flattening in practice should surround such designs.

2.2 Empirical Results

Two examples demonstrate the ability of the flattener to improve runtime performance. (The source code is shown in appendix A.)

Example 1 The base class has three virtual functions, that increment a member variable by one, two, and three respectively. A derived class overrides the first two to decrement rather than increment. A function accepts by reference two objects of nominal base type, and invokes the three methods on each ten million times. The driver invokes the function with two derived objects.

Example2 This is a set of matrix classes similar to those described in [3]. An abstract base class declares pure virtual methods for determining the number of rows and columns, and also for element extraction. A derived class implements these methods for a row-major matrix representation. The driver program computes the sum of all elements in a 2000 X 2000 matrix.

These examples are among the best cases for flattening:

- The virtual functions are short expressions or manipulations on member variables.
- The functions are readily inlinable.
- Inlining leads to further compiler optimizations.
- The most common client applications involve derived class objects used in ways that do not require dynamic polymorphism.
- Example2 is a classic abstract class based design where superclasses exist to define interfaces, not implementations, forcing *all* functions to be defined virtually solely for declaration effects.

2.2.1 Timings

The following timings were observed on an otherwise idle HP/720 workstation using the HP CC compiler.

File	Time (secs)	Flat Time (secs)	Speedup (%)	Slowdown (times)
Example1	66.3	11.88	82.10	5.58
Example2	8.22	1.93	76.52	4.25

$$Speedup = 1 - (FlatTime/NonFlatTime) \quad (1)$$

$$Slowdown = NonFlatTime/FlatTime \quad (2)$$

These figures demonstrate the benefits of flattening; for example, the flattened Example1 executes 82.1% faster than the original. Slowdowns, describe these from the

opposite viewpoint, in terms of the "penalty" for declaring functions virtual; for example, introducing virtualness in Example1 causes it to run 5.58 times more slowly.

To demonstrate that this pattern of effects is generally machine and compiler independent, the following results were collected on a Sun Sparc1+ workstation using the GNU g++ compiler:

File	Time(secs) (secs)	Flat Time (secs)	Speedup (%)	Slowdown (times)
Example1	66.5	23.20	64.00	2.83
Example2	8.34	3.66	56.11	2.27

Flattening can only provide backend optimizers with the raw material for further program analysis. It was apparent while running these examples that extremely diligent compilers might have further optimized these programs. For example, some array index bounds checks that could be statically deduced to never trigger were not removed by either compiler. As an experiment, a carefully hand optimized version of the matrix sum function in Example2 was written, and found to have execution time of 0.43sec on the HP720 showing that speedups of up to 97.7% (slowdown factor 44.2) might in principle be attainable with flattening if compilers fully exploited all of the opportunities flattening may provide.

On the other hand, flattening is typically useful only for specially selected "bottleneck" classes and functions. The effects of flattening on larger programs in which these components play lesser roles will be less dramatic.

2.3 Program Understanding

One of the obstacles to reuse in programming environments is that it is often extremely difficult to just locate all the classes that one needs to use [5]. Extensive use of inheritance can make it more difficult to track down all the methods of a class derived from several parents. Class browsers with the flattening option will allow a developer to view members of the base classes in addition to the class being investigated.

Developers have often complained about the difficulty of tracing the execution of code with inheritance, since invocations often flip back and forth between base and derived class definitions of methods [6]. Debugging flattened code is simpler due to the lack of a hierarchy; execution of the program conforms with the intuitive view of the programmer. For example:

```
Class A {
```

```
    public:
         void a() { /* code for A::a */ }
         virtual void f() { /* code for A:f */ }
}

class B : public A {
    public:
         void b() { /* code for B::b */ }
         void f() { /* code for B::f (overriding A::f) */ }
}
```

can be flattened into a stand alone, single class something like:

```
class FlatB {
     public:
         void a() { /* code for A::a */ }
         void b() { /* code for B::b */ }
         void f() { /* code for B::f (overriding A::f) */ }
}
```

A programmer who wants to reuse `class B` now need look only at `FlatB` rather than go through both `class B` and `class A`. The flattened class also has no virtual functions and is a stand alone class (base class). Any uses of `class B` type objects will now be flattened to `FlatB`. For example `void foo(B& b);` will be flattened into `void foo(FlatB& b);`

3 Design of a Class Flattening Tool

The process of flattening may be divided into three separate issues, described in more detail below.

Class Flattening: Creation of the principle flattened class(es).

Variable Flattening: Substitution of selected variable declarations and object constructions with flattened versions.

Client Function Flattening: Creation of functions accepting flattened versions of classes as arguments, corresponding to each original function accepting the original classes.

The design of a flattener is driven by the necessity to parse, store, transform, modify, and output C++ code without altering the semantic structure in any way. It shares a number of assumptions held in common across many such tools:

1. The original code passes successfully through a C++ compiler. Therefore, the flattener itself does no *checking*.
2. Since relevant declarations and usages may occur within any file of a multi-file program, the *entire* program is put through the flattener at once. Alternatively, a programmer may submit a segment of a program known to be entirely self contained with respect to the flattened classes. (Note however that if source code for any relevant class or method definition is not available, the tool may fail.)
3. Unless otherwise directed, the flattener may only *add* new declarations and definitions, not alter existing ones.

3.1 Flattening Classes

There are two principle design alternatives for approaching the elimination of virtual function calls from class hierarchies. The first is to create a stand alone class for each derived class, that encapsulates all of the functions and data of the classes within the hierarchy. We will refer to this alternative as the *stand-alone* approach in subsequent discussions. The second path is to create a second class hierarchy, paralleling the original, but in which all virtual functions are transformed into non-virtual ones. We will refer to this approach as the *parallel-hierarchy*.

3.1.1 Stand-Alone Classes

A stand-alone class is a flattened class which is not a part of the inheritance hierarchy of the program. An indicated (derived) class is used as the basis for constructing a (co-existing) flattened version. For each selected class, a new flat class is created by including all relevant members of each of the base classes. This inclusion is done recursively by

considering the parents of each of the immediate parents and so on until the flat class being formed is truly stand-alone.

The essence of flattening is a weak form of symbolic execution. All operational class relations, invocations, etc., in the original version must appear in the flattened version. Because of the large number of C++ constructs available to define subclasses and their relations to others, there are a large number of corresponding issues, including the following:

Name mangling: Since it is possible for both the base and derived classes to have identically named members (data and methods) we need to "mangle" the names of the base class members included in the flat class, and to track these names throughout all other methods. For example:

```
class A{
      int x;
      void m() { ++x; }
};

class B : public A{
      int x;
};


class flatB{
      int x;
      int FlatAx;
      void m() { ++FlatAx; }
};
```

Type Of Inheritance: There are three ways in which a class can be used as a base class for a derived class:

1. *Public:* No special actions.
2. *Private:* All inherited data and functions are retained. Even though inherited members need not be not accessible from clients of the derived class, they may still be invoked internally.
3. *Virtual:* The data representations of common bases must be included exactly once.

Constructors and Destructors Both constructors and destructors are not inherited at all. But to deal with situations wherein the constructor of a base class is called in the initialization list of the derived class constructor we need to include the constructors of the base class in the flat class. One way to do this is by treating the constructors as normal methods in the flat class. Other initializations (e.g., refer-

ences and constants) in the base must be transfered to the initialization sections of the flattened derived. For example:

```
class A {
   public:
      int x; const int y;
      A(int m, int n) :y(n) {
        x = m;  }
      A() { }
      ~A(){A's destructor code;}
};
```

```
class B : public A{
  public:
    B(int i, int j) : A(i,j){ }
   ~B() {B's destructor code;}
};
```

```
class flatB {
   public:
      int FlatAx; const int FLatAy;
      FlatAA(int m, int n) { FlatAx = m; }
      B(int i, int j) :FlatAy(j) { FlatAA(i,j);}
      ~B() {
           B's destructor code;
           A's destructor code;
      }
};
```

As here, we *do not* include the destructors of the base class in the flat class. since these members are never explicitly called for the base class. However the nature in which they are called implicitly requires that for a stand alone class the destructor code of the base class be included *after* the destructor code of the derived class in the destructor of the flat class. In case of multiple inheritance it is required to add the destructor code of the base classes in the reverse order of their declaration as parents of the derived class.

Propagated Declarations Typedefs and enumerations in the base class are inherited and have to be included in the flatclass. But they need not be mangled in any way.

Static Methods Operator `new` is a good example. It is inherited, but the inherited form can only be invoked using the scope resolution operator. Since it cannot be included as `operator new` it must be included as a new member function of the flat class rather than as an operator. (Other operator-syntax methods require similar treatmnent.) For example:

```
class base {
     public:
```

```
        void* operator new(size_t t) { ..... }
        ...
};

class derived : public base {
      public:
        void* operator new(size_t t) { ..... }
};

class flatderived {
      public:
        void* operator new(size_t t){ ... } // Derived class's version.
        void* Flatbasenew(size_t t){....}   // Base class's version.
};
```

Friends Due to our assumption that the code input into the tool doesn't have any access violations, "friends" declarations are made unnecessary by making all data members and functions public. Thus, they may safely be ignored.

Self Clients Methods and operators that take other objects of the same class as arguments are special cases of client functions, discussed below.

Assessment. The main problem with the stand-alone approach is its extreme sensitivity to details in C++ "static semantics". While this need not be a difficult issue in itself[1], it is a serious limitation that the tool cannot simply adapt to changes in the language and refinements in compilers as C++ becomes standardized.

3.1.2 A Parallel-Hierarchy of Classes

The parallel-hierarchy approach to flattening classes addresses some of the problems associated with the stand-alone approach. A parallel hierarchy may be generated to co-exist with an original hierarchy. Each class differs from its corresponding original class *only* in that all virtual functions are made non-virtual. Of course, the names of the classes are appropriately adjusted to reflect that they are flat classes. For example, given the following hierarchy:

```
class A {
public:
virtual f();
```

[1]But usually is. Given that no two existing C++ compilers currently agree on their interpretation and implementation of some such details, the chances that a separate tool would be compatible with any one of them appears remote.

```
virtual g();
h();
};

class B : public A {
public:
virtual f();
virtual g();
h();
};
```

This would be flattened into a co-exisisting parallel hirarchy:

```
class FlatA {
public:
f();
g();
h();
};

class FlatB : public FlatA {
public:
f();
g();
h();
};
```

Assessment. This approach does *not* address class browsing applications, since it does nothing to move all defintions to a single view. Another deficiency is that many additional intermediary classes must be generated. This can add to the "code bloat" already associated with the tool.

However, this approach is significantly better for flattening for performance. The most important design advantage is simplicity. All subclass, resolution, and access control rules in C++ work the same whether member functions are declared as virtual, non-virtual, and/or inline. Thus, the flattener itself need not even be aware of such rules, at least for purposes of flattened class generation. It may simply rely on the downstream compiler to handle these issues. This also enhances maintainability. The flattener need not be reworked to comply with evolving C++ semantics rules.

3.2 Flattening Variables

3.2.1 Using Flat Classes

Regardless of which of the above approaches is chosen, there are three ways of using flat classes in other parts of a program. The first method is to flatten some classes and use these flat classes with knowledge about how the flattener produced them. Programmers can, for example, flatten a specific derived Matrix class and begin using that flattened class in their new code (for speed). The drawbacks of this approach are that an intimate knowledge of how the flattener works is required, changes in the code in any part of the Matrix hierarchy won't be automatically propagated to the flattened classes. Also, changes in flattener requires that the programmer be up to date on the way it works, etc.

The more interesting way is the automatic method. In this method, the flattener goes through the source code of a program and changes the type of certain variables from "Type" to "FlatType". Only a restricted set of variables can be safely converted to use flat classes. The disadvantage of this approach is that without prohibitively expensive analysis, the flattener has to be conservative about what it can flatten. There are some cases where the flattener might be able to use flat classes but doesn't because the code broke one of the (simple) rules that the flattener didn't consider safe, even though a human might be able to determine that it is safe. Of course, the automatic method has the obvious advantage thatprogrammers do not need to know how the flattener works; code can experience a speed-up as a result of blindly using the flattener.

The last method is a combination of the first two. The flattener performs certain substitutions automatically, but programmers are allowed to sprinkle source code or otherwise indicate directives to the flattener. For purposes of generating a practical tool, this appears to be the best alternative.

In the remainder of this section, we assume use of the parallel-hierarchy approach with respect to method names, etc.

3.2.2 Criteria

It is possible to define various criteria for dealing with variables of flat classes instead of their corresponding classes in programs (global variables), functions (local variables), and classes (class members). It is generally safe to flatten declarations of the forms:

1. `ClassName var1;`
2. `ClassName& var2 = x;` where x is of the flattened type.
3. `const ClassName* var3 = y;` where y is an address of the flattened type.

In all these forms, the types of the variables are known and fixed. That is, var1, var2, and var3 are all bound to objects whose types are fixed while the variables exist. Since the types are known and fixed, there is no need to use the dynamic dispatching facilities of C++.

It is not always possible to flatten regular pointers to objects (ClassName* var4 = ...) because different objects may be bound to those pointers at different times during the execution of the program. If it were feasible to do a full data flow analysis of the entire C++ program, it would then be possible to flatten pointers which are known to only point to a certain type of object throughout the program's execution (that is, the pointer isn't used polymorphically). Of course, there is a very high cost associated with performing this kind of static analysis (if it is possible at all).

3.2.3 Failures

If the address of a potentially flattenable object is taken (directly, by using the & operator; or indirectly, by assigning it to a reference or by passing it by reference to a function), the object cannot *always* be flattened. Although the object's type will not change, the pointer to the object may be assigned to more basic pointers. In that case, since the flattened object is not in the inheritance hierarchy anymore, this would cause a type conflict and will make the C++ compiler fail.

Consider this code fragment, assuming base class `A`, subclass `B`, and flattening on `B`:

```
f() {
  B d; // the flattener would make this statement ``FlatB d;''

  B* dp = &d;
  B& dr = d;

  A* bp = dp; // this is allowed by C++
}
```

In this small example, `d` should not be flattened. The assignment "`A* bp = &d;`", where `d` had been flattened, would not be allowed by the C++ compiler (since `d`'s flattened type `FlatB` is not derived from `A`). Changing `bp` to match the type of `d` is not allowed. This is because `bp` might later be used to hold other derived objects of `A`.[2]

This analysis is difficult to automate. Once all of the variables are pre-screened (using the preliminary selection criteria discussed above), it is then necessary to examine the usages of those variables in the scopes in which the variables are accessible. For global variables, the entire program must be examined. For class member variables all methods, friend functions, and methods of friendly classes must be examined. For local variables, only the function containing the local variable needs to be examined.

If one could trace the use of the variables passed by reference or pointer, more constructs would be automatically flattenable. But because the most conservative rules

[2] An alternative strategy would be to automatically define conversion operators between flattened and original classes. However, this would almost always signficantly change the semantics between original and flattened versions of the program.

fail in many, many common situations, explicit indication of to-be-flattened variables is a better alternative.

A few intrinsic safeguards help make manual indication of flattenable variables less dangerous than it might be otherwise. In many cases, the tool will generate illegal C++ code (caught as such by a downstream compiler) when flattening is not possible. For example, if a programmer erroneously indicated that `d` in the above example were flattenable, the illegal code generated by the `A* bp = dp` statement would cause the flattened program not to compile. (This fact suggests a naive, time-consuming, but generally effective tuning strategy for users: Start by assuming that all object declarations are flattenable, and then revise downward (or perhaps revise overly broad pointer and reference variable declarations) on compilation failures.)

3.3 Client functions

A "client function" of a flattened class is any function that takes as an argument any instance of the flattened class or one of its ancestors. An important special case of a client function is a client member function that takes such arguments. Client functions are those where almost all virtuality-based performance hits actually occur, since they are the sites where most virtual calls would otherwise be made.

Client functions may themselves be flattened via duplication, using the same basic idea used in parallel-hierarchy generation. For example, assuming the declarations in the previous section, if there were originally:

```
void c(A* a) { a->f(); a->g(); }
void d(B* b) { b->f(); b->h(); }
```

Assuming flattening on class `B`, flattened versions of clients may be *added* to the code stream:

```
void c(FlatB* a) { a->f(); a->g(); }
void d(FlatB* b) { b->f(); b->h(); }
```

In these examples, nothing at all was altered except the function declarations themselves. All references to `B`'s and ancestors thereof were converted to `FlatB`'s.[3] As with parallel-hierarchy generation, the original versions coexist in the code stream. C++ function call resolution mechanics within a C++ compiler will choose the appropriate version.

Functions with multiple arguments of relevant classes may require multiple flattened versions, corresponding to all ways in which the flattened version may occur.

[3] In a stand-alone approach, other name-mangling conventions must also be adhered to.

3.3.1 Local Variables

When client functions include declarations of new variables, a combination of the duplication and variable-flattening strategies must be employed. But here, variable-flattening need not be so conservative. A simple 'universal demotion' strategy is attractive. Here, *all* occurences (variables, objects) of the original class or its ancestors are transformed into the flattened version. For example:

```
void e(A* a) { A* p = a; p->f(); p->g(); }
```

transforms to

```
void e(FlatB* a) { FlatB* p = a; p->f(); p->g(); }
```

Demotion does not extend to scope designators. For example, any occurence of `A::f()` inside a flattened`B` is simply converted to `FlatA::f()` (using the parallel-hierarchy method) or `FlatAf()` (using the stand-alone method).

3.3.2 Limitations

As mentioned above, flattening is generally most attractive and effective when superclasses are *abstract*. In these cases, since abstract classes may not be instantiated, client functions will not include constructions of superclass objects, and no further considerations apply.

But uniform demotion *can* change semantics in the case where superclass *objects* are created within client functions. For example:

```
void v(A* a) { A x; x.f(); x.g(); }
```

transforms to

```
void v(FlatB* a) { FlatB x; x.f(); x.g(); }
```

The object `x` was a base object in the original, but a derived object in the transformed code. The effects of this transform are always *safe* and not incorrect, but cannot be guaranteed to be *desired*. While experience shows that these modifications produce code deemed acceptable by programmers, they do reflect a certain arbitrariness owing to the fact that there is no way to automatically assess desirablity. Because failure to generate flattened code in these situations would substantially limit the utility of the tool, the only pragmatic solution is to make these transformations anyway. Users of a flattening tool must be informed that these kinds of modifications will occur and/or be allowed to prevent them on a case-by-case basis. Many further refinements are possible.

4 Project History

The current prototype version bears many scars from its development history, as described in this section.

4.1 Parsing

In summer/fall 1991, the flattener was undertaken as one of several projects that could be used in order to simultaneously gain experience with relatively "shallow" preprocessor-based (annotated C++ code in, pure C++ code out) tools while also producing software of practical value.

Initial efforts surrounded examination of existing C++ parsers and related front-end utilities. We chose to base these upon a yacc-based grammar designed by James Roskind.[4] At the time, the reasons appeared compelling:

- Developing our own parser would have taken too long, and diverted efforts from tool building to parser building.
- The grammar accepted most constructs then accepted by most C++ compilers. No other available choice came as close.
- Several other ad hoc tools based on this grammar were in development elsewhere.

In retrospect, this was not the best decision. The grammar is not readily extensible, and has not been updated to reflect the range of constructs currently accepted by most compilers (including especially templates).

Also, most early work surrounded the construction of minimal support structures required by any such tool. Because of the limitations of yacc-based LALR actions (even within allegedly C++-friendly modifications), most internal programming could only use aspects of C++ that remained close to C. We have often witnessed firsthand the well-known differences in extensibility of procedurally-based just-barely-C++ code surrounding the parser and the few parts of the tool that have followed standard OO design principles.

4.2 Stand Alone Design

We first tried a version that could serve both browsing and optimization roles, using the stand-alone strategy, along with highly conservative variable-flattening rules. This design was partially completed. (It successfully passed a number of small test cases.) It was reported upon at the C++ At Work Technical Sessions in November 1991.

However we became increasingly concerned about some of the limitations noted above. To recap and emphasize:

[4]We gratefully acknowledge the help given to us by James Roskind.

1. The tool was becoming too intimately tied with C++ semantic details. Maintaining total correctness across the range of possible C++ constructs was becoming too expensive for an allegedly "lightweight" tool. Too much effort was involved in attaining correctness across the range of all possible C++ constructs. Repairing this in the right way would require a commitment to a better internal representation of semantics, which is itself a controversial area.
2. Increased coupling with detailed semantics makes the tool difficult to maintain and to evolve so as to correspond to changes in standard compilers and to the language.
3. Conservative variable-flattening rules caused the tool to fail to effectively flatten many (actually most) common designs.

4.3 Parallel Hierarchies

The above version was modified during spring/summer 1992 to implement the basic features of the parallel hierarchy design. As described above, this modification is more successful in decoupling the tool from C++ details. This substantially enhances output code correctness, compatibility and maintainability.

We simultaneously began revising some of the support for our revised strategies involving manual indication of variable-flattening in conjunction with a slightly more agressive client-function variable rule (uniform demotion). These eliminate several limitations in the previous version at the expense of manual intervention.

Together, these modifications have led to a fairly successful model of how to construct a simple, usable flattening tool.

Unfortunately, the current version does not well-reflect these properties. Between the difficulties of dealing with the `yacc` grammar and surrounding support code, and the fact the many internals were radically altered between versions (as well as numerous fumblings while in the midst of each), the current code must be considered in the same light as most other "experimental" software efforts. While it successfully passes some test cases, it is not of production quality.

Also, the current version does not implement all of the described design features. Most of conservative variable substitution rules implemented in the stand-alone version are still used, and the tool is not fully interactive. It uses some `pragmas` to describe class flattening targets, and does not support manual indication of variables to flatten. It currently passes only those tests that do not require intervention or full local variable demotion rules.

The main rationale for leaving the prototype in this state is that it has served its purposes in revealing the utility, design options and principles, and feasibility of flattening. Like many other recent efforts to build C++ tools, our experiences have shown that is surprisingly difficult and unproductive to construct systems that take it upon themselves to parse and represent C++ source code, even for simple purposes. Tools like this become practical only when built upon a common infrastructure performing these tasks.

5 Recommendations

5.1 Flattening Tools

Construction of a production-quality version of a flattening tool would involve a series of policy and implementation issues.

5.1.1 Policy Issues

The goal of producing browsable flattened versions of classes should be separated from that of performance improvement. Browsable versions should be produced via different tools or toolsets that emphasize readability over adherence to detailed semantics. Given this, there seems no question that the parallel-hierarchy approach should be adopted for class flattening.

The tool should be interactive, and have a full menu-driven graphical interface. This will allow programmers to more simply and effectively indicate which classes and especially which variables/objects to flatten.

The tool should either use a well-maintained front-end that forms a common basis for a variety of tools, or become integrated into a compiler itself. Given the unlikelihood of interactive compilers, the former appears to be the only serious option for a tool (but see below about other options.)

No attempt to further automate detection of flattenable entities should be made until compilers or static analyzers with full high-level dataflow capabilities are constructed. However, variants and improvements of local variable handling strategies within client functions should be further explored.

5.1.2 Implementation Issues

Re-basing the tool on an existing syntax and semantics framework would radically simplify internal design. The vast majority of the code in the current version has little to do with flattening. In fact, we believe that the current parsing and syntactic representation code should simply be scrapped and replaced with an interface to a multipurpose semantically oriented front-end (or server).

The internal architecture of any such front-end would govern most design details. However, with such links, there remains only the support for the three basic transformation functions within the flattener proper:

1. Echoing flattened versions of each class in a hierarchy.
2. Generating flattened client functions.
3. Recasting variable declarations and object constructions into flattened form.

(Recall that this is a functional description, not a design; also recall the three cases may interact, e.g., in the case of client member functions appearing in flattened classes.)

The basic requirements for (1) and (2) are remarkably similar to those for dealing with C++ `templates`. Definitions must be stored and then output in transformed form. It appears very likely that any available existing mechanisms that perform template expansion could be generalized to also perform flattening.

5.2 Flattening Within Compilers

If the constraints that lead flattening to be performed within a separate tool are relaxed, different options become available for providing at least some of the performance enhancment functionality of flattening within C++ compilers.

As described in section 2, the basic idea of flattening is to arrange that `virtual inlines` actually get inlined. A flattening tool transforms C++ code in ways that allow standard C++ compilers to do just that. Construction of flattened classes and variables is the best (perhaps only) path to this. However the real improvements come within client function code. Flattened clients are presented to a compiler in such a way that there is no compile-time uncertainty about which version of a function to call.

There are other routes to uncertainty reduction that could be pursued within a compiler rather than a tool. The most feasible strategy appears to be *leaf class* customization. Leaf classes are those classes that have no subclasses. They are, by far, the most common targets for flattening. They are also mechanically discernible.

The idea is to

1. Detect leaf classes.
2. For each of these, generate flattened versions of each original client function accepting them as arguments (as in section 3).
3. Within each flattened client function, consider all virtual calls to be preresolved to the target leaf class. (The same rule applies to all self invocations within all methods of the leaf class.)
4. For each invocation of a flattened client function, insert code to (dynamically) dispatch to flattened versions if the arguments match the special versions.

The main logistic problem is in keeping track of leaf classes, their ancestries, and clients. Since these could appear in almost any order within the source text, and may extend across compilation units, this appears feasible only if such optimizations were performed only when this information were made available to a compiler *before* scanning source code (e.g., via tool and environment support in the style of the previous section.)

Step (4) is made practical only if the compiler already supports some form of run-time type information scheme. Dispatching to flattened clients in the case of leaf classes may be performed via 'type tests' that invoke flattened versions only if the classes are indeed the target leaf classes. Note that within clients, invocations of other flattened client functions need not be conditionally dispatched for objects whose type is fixed within argument lists.

This framework need not deal with the local variable problem within client functions to remain an effective optimization strategy. Locals within clients will not necessarily always enjoy all the results of flattening. However, in practice, these cases do not appear to represent the bulk of real bottlenecks.

Anything beyond this appears infeasible in C++. Flattening of "intermediate", non-leaf classes would require much additional analysis, and would provide diminishing returns with respect to performance since, as mentioned above, most flattening opportunities involve leaf classes.

A Source Code for Example 1

```
#include <stream.h>
#include <builtin.h>

class Base{
  protected:
    int size;
  public:
    Base(){}
    virtual void vf1(){size++;}
    virtual void vf2(){size+=2;}
    virtual void vf3(){size+=3;}
};

class Derived:public Base{
  public:
    Derived(){}
    void vf1() {size--;}
    void vf2(){size-= 2;}
};

void test(Base& d1, Base& d2)
{
  for(int j=0; j < 10000000; j++)
  {
    d1.vf1();
    d2.vf1();
    d1.vf2();
    d2.vf2();
    d1.vf3();
    d2.vf3();
  }
}

main()
{
  double t;
  Derived d1;
  Derived d2;

  // timer is used to compare the timings for the
```

```
  // original versus flattened versions of this program
  start_timer();

  test(d1, d2);

  t = return_elapsed_time(0.0);
  cout << "Time = " << t << "\n";
}
```

B Source Code for Example 2

```
#include <stream.h>
#include <builtin.h>

class Matrix
{
public:
  virtual       ~Matrix() {}
  virtual int    rows() const = 0;
  virtual int    cols() const = 0;
  virtual float  elem(int i, int j) const = 0;
  int            size() { return rows() * cols(); }
};

float sum(const Matrix& m)
{
  float s = 0;
  for (int i = 0; i < m.rows(); ++i)
    for (int j = 0; j < m.cols(); ++j)
      s += m.elem(i, j);
  return s;
}


class DenseMatrix : public Matrix
{
public:
  virtual float&   operator () (int i, int j)  = 0;
};

class RowMajorMatrix: public DenseMatrix
{
private:
  friend float fastsum(const RowMajorMatrix&);
  int r; int c; float* d;
public:
         RowMajorMatrix(int m, int n) :r(m), c(n), d(new float[m * n]) {}
        ~RowMajorMatrix() { delete d; }
  int    rows() const { return r; }
  int    cols() const { return c; }
  float& operator () (int i, int j)
```

```
        {
          if (i < 0 || i >= rows() || j < 0 || j >= cols())
            abort();
   else
                ;
          return d[i * cols() + j];
        }
  float  elem(int i, int j) const { return (*((RowMajorMatrix*)(this)))(i, j); }
};


float fastsum(const RowMajorMatrix& m)
{
  float  s = 0;
  float* p = &(m.d[0]);
  float* fence = &(m.d[m.r * m.c]);
  while (p < fence) s += *p++;
  return s;
}

#define N 1000

main()
{
  RowMajorMatrix m(N,N);

  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j)
      m(i, j) = 1.0;

  double s;
  double t;

  start_timer();
  s = sum(m);
  t = return_elapsed_time(0.0);
  cout << " sum: ";
  cout << "sum = " << s << " Time = " << t << "\n";

  start_timer();
  s = fastsum(m);
  t = return_elapsed_time(0.0);
```

```
  cout << " fastsum: ";
  cout << "sum = " << s << " Time = " << t << "\n";

}
```

References

[1] D.Hahn B.Cohen and N.Soiffer. Pragmatic Issues in the Implementation of Flexible Libraries for C++. In *USENIX C++ Conference*, 1991.

[2] Eli Charne. Lessons learned implementing a browser for c++. In *C++ At Work Conference*, 1991.

[3] Doug Lea. Customization in C++. In *Proceedings Usenix C++ Conference*, 1990.

[4] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages practica. In *Proceedings OOPSLA*, 1991.

[5] Mary Fontana and Martin Neath. Checked out and long overdue: Experiences in the design of a C++ class library. In *USENIX C++ Conference*, 1991.

[6] Scott Meyers. Working with object-oriented programs: The view from the trenches is not always pretty. In *Symposium on Object Oriented Programming Emphasizing Practical Applications*, 1990.

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1991.

[8] M.Ellis and B.Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990.

[9] Stanley B Lippman. *C++ Primer*. Addison Wesley, 1989.

[10] James Roskind. A YACC-able C++ 2.0 grammar, and the Resulting Ambiguities. (Publically available software.).

[11] David S. Rosenblum and Alexander L. Wolf. Representing Semantically Analyzed C++ code With Reprise. In *Proceedings, USENIX C++ Conference*, 1991.